

# The Gaming Room Web Application

# **CS 230 Project Software Design Document**

Version 1.2

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 4**](#_Toc115077323)

[**Domain Model 4**](#_Toc115077324)

[**Evaluation 6**](#_Toc115077325)

[**Recommendations 7**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 03/18/25 | Sulakshana Rane Mehta | Initial version of the software design documentation, outlining the system architecture, requirements, and recommendations for expanding 'Draw It or Lose It' into a web-based application. |
| 1.1 | 04/04/25 | Sulakshana Rane Mehta | Revised to align with CS 230 Project Two requirements: Added evaluation of platforms (Linux, Mac, Windows, Mobile) considering server and client-side aspects. |
| 1.2 | 04/18/25 | Sulakshana Rane Mehta | Added detailed recommendations per Project Three requirements: including operating platform selection, system architecture overview, storage and memory management, distributed system connectivity, and cross-platform security protocols. |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room is looking to expand its popular Android-based game, **Draw It or Lose It**, to a web-based platform that supports multiple operating environments. The objective is to design a scalable, distributed software application that allows multiple teams to compete while ensuring unique team and game identifiers.

This document also outlines the key software requirements, design constraints, and system architecture, along with a detailed evaluation of different operating platforms to determine the best environment for development and deployment.

By leveraging object-oriented programming principles and adhering to design constraints, the proposed solution will provide a seamless gaming experience across different platforms.

## Requirements

The key business and technical requirements for the new gaming application should include:

* Multi-team functionality, where each game session can have multiple teams.
* Multiple players per team with the ability to assign and manage them dynamically.
* Unique naming constraints for teams and games to prevent eventual duplication and allow clear identification.
* Single-instance enforcement to ensure only one instance of a game exists in memory, requiring unique identifiers for game elements.
* Compatibility with various operating systems (Windows, Linux, Mac) and mobile platforms for scalability.
* Real-time rendering of drawings within a fixed timeline (30 seconds for complete rendering, 15-second additional guessing time for other teams).
* Secure user authentication and game state management to prevent data inconsistency.

## [Design Constraints](#_2et92p0)

Developing a web-based, distributed gaming application introduces several constraints:

* **Performance Constraints:** The game must render images in real time, necessitating efficient data streaming, optimized rendering pipelines, and low-latency networking. Techniques such as edge caching and GPU acceleration should be considered to reduce lag.
* **Scalability Constraints:** The application should support multiple concurrent players and teams while maintaining responsiveness. Load balancing techniques, microservices architecture, and cloud auto-scaling should be used to handle peak traffic efficiently.
* **Concurrency and Synchronization:** Ensuring synchronized updates across different users and devices is crucial. The application must implement real-time messaging protocols such as WebSockets or gRPC, along with conflict resolution mechanisms to handle simultaneous interactions.
* **Security Considerations:** Data transmission between clients and servers must be secured to prevent cheating, unauthorized access, or data leaks. Implementation of end-to-end encryption (SSL/TLS), secure authentication (OAuth 2.0, JWT), and anti-cheat mechanisms will enhance security.
* **Cross-Platform Compatibility**: The game should function seamlessly across different operating systems and devices without requiring any platform dependencies. Cross-platform development frameworks such as Unity, React Native, or WebAssembly should be leveraged to ensure consistent user experience.
* **Memory Management**: Efficient memory allocation strategies should be used to manage concurrent users without excessive memory usage. Techniques such as garbage collection tuning, object pooling, and memory profiling should be incorporated.
* **Network Reliability**: The application should handle intermittent connectivity issues and ensure game state persistence. Implementing offline mode with local caching, automatic reconnection mechanisms, and data synchronization strategies will improve reliability.

## [System Architecture View](#_ilbxbyevv6b6)

Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

## [Domain Model](#_8h2ehzxfam4o)

The UML class diagram provided outlines the object-oriented structure of the game. Key components include:

* **Game Class**: Represents the overall game instance, responsible for managing teams, handling game rules, tracking progress, and ensuring smooth gameplay coordination. It serves as the primary controller for the game lifecycle.
* **Team Class**: Represents a team within a game session, storing team-specific details such as a unique identifier, assigned players, and team name. It facilitates team-based interactions and maintains team organization within a game.
* **Player Class**: Represents individual participants within a team. Each player instance contains attributes such as a unique identifier, name, and potentially other gameplay-related statistics. It ensures that players are properly tracked within the game.
* **Entity Class**: A base class designed to encapsulate shared attributes across different game objects, such as unique identifiers, creation timestamps, and metadata. This promotes reusability and maintains a clean object-oriented design.
* **GameService Class**: Implements the Singleton design pattern to ensure only a single instance of game management exists throughout the application. It provides methods for game creation, retrieval, and assignment of unique game and team identifiers, enforcing centralized game state management.

**Relationships**:

* The Game class has a **one-to-many relationship** with the Team class, meaning each Game can involve multiple Teams, with each Team participating in a specific Game. Furthermore, each Team class has a **one-to-many relationship** with the Player class, indicating that one Team can have multiple Players, each playing for that Team in various Games.

**OOP Principles**:

* **Encapsulation**: The game data, such as player details and team information, is stored within classes, with controlled access through getter and setter methods. This ensures data integrity and prevents unintended modifications by external components.
* **Inheritance**: The Entity class acts as a parent class, providing a common structure for other classes like Player, Team, and Game. This promotes code reuse and maintains a hierarchical structure, reducing redundancy in the codebase.
* **Polymorphism**: Different game components can have their own implementations of shared behaviors. For example, the Team class might override a method from the Entity class to customize how team data is handled, ensuring flexibility in extending game functionalities.
* **Singleton Pattern**: The GameService class follows the Singleton design pattern, ensuring that only one instance of the game service exists throughout the lifecycle of the applicaton. This prevents duplicate game sessions, maintains consistency in game state management, and optimizes resource utilization.

**"The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.**

## [Evaluation](#_2o15spng8stw)

The following tables present an evaluation of the advantages and weaknesses of Linux, Mac, Windows, and Mobile platforms, divided into Client-Side and Server-Side considerations. There’s also a brief overview of recommended development tools.

## Client-Side Platform Evaluation

|  |  |  |
| --- | --- | --- |
| Platform | Advantages | Weaknesses |
| ****Linux**** | - Open-source and free - Highly customizable - Lightweight, good for older devices | - Limited support for commercial games and applications - Higher learning curve for non-technical users |
| ****Mac**** | - Strong graphics performance - Secure environment - Smooth user experience | - Expensive hardware - Limited upgrade options - Smaller gaming market share |
| ****Windows**** | - Widest hardware compatibility - Largest user base for games - Extensive support for drivers and applications | - Higher vulnerability to malware - Frequent system updates may disrupt gameplay  - Licensing costs for Windows OS and development tools can be high |
| ****Mobile**** | - Native app access (iOS, Android) - Always available and portable - Touchscreen enhances game interaction | - Limited processing power compared to desktops - Fragmented OS versions across devices can make consistent app performance - Smaller screen sizes may affect game visibility |

## Development Tools

## For client-side development across various platforms, the recommended tools and languages differ to best match each system’s environment and user expectations.

## On Linux, developers can utilize languages such as Java, Python, and C++. Popular integrated development environments (IDEs) like Eclipse, NetBeans, and Qt Creator provide robust support. Frameworks like GTK+ and SDL are ideal for creating graphical user interfaces and rendering real-time drawings efficiently.

## For Mac, development is typically done using Swift, Objective-C, and C++. Xcode is the primary IDE, offering seamless integration with macOS APIs. Frameworks like Cocoa and Metal help optimize graphics performance, which is critical for a visually intensive game like Draw It or Lose It.

## On Windows, developers frequently work with C#, C++, and Java. Visual Studio serves as the main IDE, with additional support for gaming engines like Unity and Unreal Engine. Key frameworks such as .NET and DirectX assist in building high-performance, graphical applications suitable for mass-market distribution.

## For Mobile platforms, development typically splits between Android and iOS ecosystems. Kotlin and Java are preferred for Android development using Android Studio, while Swift is used for iOS development within Xcode. Cross-platform frameworks like Flutter and React Native can also be employed to maintain a single codebase across mobile devices, improving efficiency.

## Server-Side Platform Evaluation

|  |  |  |
| --- | --- | --- |
| ****Platform**** | ****Advantages**** | ****Weaknesses**** |
| ****Linux**** | - Most widely used for servers - Open-source, cost-effective, highly customizable - Stable and reliable for continuous hosting - Strong community support and frequent security updates | - Requires skilled administration - May lack official support for some enterprise tools - Command-line management can be challenging for beginners |
| ****Mac**** | - Secure and stable Unix-based system - Excellent environment for development and testing - Native support for Docker and CI/CD workflows | - Expensive hardware - Rarely used for production hosting - Limited community support for server-specific needs |
| ****Windows**** | - Familiar interface for IT teams - Strong support for .NET, Active Directory, Azure - Comprehensive tools like Visual Studio and IIS | - Licensing and operational costs are high - Higher system resource usage - Less flexibility for custom server setups |
| ****Mobile**** | - Supports lightweight server tasks (e.g., notifications) - Easy integration with Firebase and APNs - Useful for peer-to-peer gaming coordination | - Not suitable for full web hosting - Limited processing power and customization - Depends heavily on external cloud services |

## 

## Development Tools

## For server-side development to support the distributed, web-based version of Draw It or Lose It, a different set of tools and languages are recommended depending on the platform.

## On Linux servers, developers typically use Python, JavaScript (Node.js), Java, or C# for backend services. Popular frameworks include Django, Express.js, and Spring Boot. Tools such as Docker and Kubernetes are recommended for containerization and orchestration to ensure scalability and performance. Apache and Nginx servers are commonly used for hosting the web application.

## For Mac servers, although less common for production environments, developers can still use Swift, Node.js, and Python effectively. The macOS environment also supports Apache and Nginx for web hosting. IDEs like Xcode and Visual Studio Code provide comprehensive development support.

## On Windows servers, C# and ASP.NET Core are the primary choices for backend development, supported through Visual Studio. IIS (Internet Information Services) is the default web server, although cross-platform tools like Node.js and Docker can also be used. Microsoft Azure integration is highly effective for cloud scalability.

## Regardless of the platform, server-side development should prioritize real-time communication technologies like WebSockets or gRPC to handle game state updates, and utilize cloud services such as AWS, Azure, or Google Cloud for scalable deployment.

## Recommendations

### ****1. Operating Platform****

To ensure the successful deployment of Draw It or Lose It across diverse computing environments, **Linux (Ubuntu Server LTS)** is recommended as the server-side operating platform. Linux offers strong performance, stability, and resource efficiency, which are all crucial for maintaining responsive game servers capable of supporting real-time multiplayer interactions. Its open-source nature allows for greater customization and control, while its lightweight footprint ensures efficient use of system resources, especially important when running numerous concurrent game sessions.

Additionally, Linux is well-supported across all major cloud infrastructures (AWS, Azure, Google Cloud Platform) and integrates seamlessly with containerization technologies such as Docker and orchestration tools like Kubernetes, making it highly scalable and deployment-friendly.

On the client side, to ensure broad accessibility and a unified gameplay experience, the game will support all major platforms: **Windows, macOS, Android, iOS**, and **modern web browsers**. This cross-platform compatibility will be achieved using a development framework such as **Unity** (ideal for 2D/3D interactive design) or **React Native** (well-suited for UI-intensive mobile apps). These frameworks enable developers to write shared code for multiple platforms, significantly reducing the cost, time, and complexity of maintaining separate native applications. This approach also ensures a consistent user interface and game behavior across devices, improving the overall player experience and supporting faster updates and feature rollouts.

**2. Operating System Architectures**

The backend operating system architecture is based on Linux's **monolithic kernel**, which offers several benefits:

* **Efficient System Calls**: All core services (file systems, device drivers, memory managers) operate in kernel space, enabling fast communication and reduced overhead.
* **Dynamic Loadable Modules**: Kernel components can be loaded/unloaded without rebooting, allowing for flexible upgrades or diagnostics.
* **Advanced Process Management**: Linux's scheduler efficiently manages multi-threaded, real-time tasks such as game state synchronization and matchmaking.
* **Scalability**: Supports vertical scaling (more resources per node) and horizontal scaling (more nodes)—critical for handling unpredictable traffic spikes in multiplayer sessions.

These features make Linux an ideal choice for maintaining performance and responsiveness under pressure, especially during peak gaming hours when multiple instances of game logic must run in parallel.

Client-side operating systems (Windows, iOS, macOS, Android) use hybrid or microkernel architectures. However, cross-platform technologies (like Unity or WebAssembly) abstract OS-specific differences to ensure a unified runtime and game logic layer. This enables consistent player experiences and simplifies development across platforms.

### ****3. Storage Management****

The game will employ a **hybrid storage architecture** that combines a relational database management system (RDBMS) such as **PostgreSQL** with **cloud-based object storage** solutions like **AWS S3** or **Azure Blob Storage** to effectively manage both structured and unstructured data.

* **PostgreSQL** will handle structured data such as user profiles, game session records, leaderboards, and player statistics. It is chosen for its **ACID compliance**, which ensures data consistency and reliability during concurrent updates, as well as for its **advanced indexing capabilities** and **extensibility**, which support complex queries and future scalability.
* **Object storage**, on the other hand, will store **large unstructured assets**, particularly the 200+ high-resolution drawing images used throughout gameplay (each approximately 8MB, totaling around 1.6GB). These files are not stored locally on user devices to conserve storage space and enhance portability across platforms. Instead, they are streamed or downloaded on demand.

As highlighted in the **CS 230 Module Six Assignment**:

*"Storage is like your filing cabinet—it holds everything for the long term... Instead of storing all images directly on every player’s device... the app can store them in the cloud and only download what’s needed."*

This approach ensures efficient resource usage by minimizing local storage demands, reducing app bloat, and allowing for dynamic content delivery tailored to each session.

**4. Memory Management**

Linux handles memory efficiently using **virtual memory, paging, and demand paging**, which allows the system to prioritize active processes and optimize RAM usage. For a fast-paced, interactive game like *Draw It or Lose It*, efficient memory use is crucial to ensure real-time responsiveness and smooth rendering. Several strategies are implemented:

* **Preloading Assets:** Before each round, **3–5 images are loaded into memory** proactively, ensuring that players do not experience lag or image loading delays during gameplay.
* **RAM Caching:** Frequently accessed elements such as **game logic scripts, session states, and user data** are stored in RAM to minimize fetch times and reduce server-client latency.
* **OOM Killer (Out-of-Memory Killer):** A built-in Linux mechanism that monitors memory usage and, during strain, **automatically terminates low-priority or idle processes** to keep the core game functions running smoothly.

As mentioned in the **CS 230 Module Six Assignment**:

*“Memory is like your desk... the game can’t afford to ‘go searching’ on the hard drive every second... instead, it should load a few images into memory before the round begins.”*

This analogy emphasizes the importance of fast, short-term memory access over slower long-term storage lookups in high-performance applications.

On the client side:

* **Unity (C#)** and **React Native (JavaScript)** environments include built-in **garbage collectors** that manage memory automatically, helping to **prevent memory leaks** during extended play sessions.
* **Temporary data**, such as real-time player drawings, guesses, or round-specific timers, is **flushed or cleared after each round**, reducing memory bloat and maintaining optimal performance on devices with limited resources.

This clear separation of concerns between volatile memory (RAM) for fast access and persistent storage for long-term asset retention is fundamental to maintaining the game’s performance, reliability, and scalability across devices and platforms.

### ****5. Distributed Systems and Networks****

*Draw It or Lose It* will operate as a cloud-native, distributed system optimized for real-time multiplayer engagement and scalability:

* **WebSockets or gRPC** will be used to maintain low-latency, bidirectional communication channels between client devices and backend game servers, enabling instant updates for drawing, guessing, and scoring.
* **Load Balancing** solutions (e.g., NGINX, HAProxy, AWS Elastic Load Balancer) will be implemented to evenly distribute incoming traffic across multiple game server instances, preventing bottlenecks during peak usage.
* **RESTful APIs** will handle stateless backend operations such as account management, leaderboard retrieval, and game history logging, allowing these services to scale independently.
* **Service Discovery and Orchestration** will be managed via Kubernetes, enabling automatic scaling, self-healing containers, and efficient service-to-service communication in the cluster.

**Networking Considerations:**

* **Secure Communication**: All data exchanges will occur over HTTPS with **TLS 1.3** to protect user privacy and prevent tampering.
* **Content Delivery Networks (CDNs)** such as Cloudflare or AWS CloudFront will be used to cache and serve static assets (e.g., game images, UI components) from edge servers worldwide, reducing latency and improving user experience.
* **Resilience and Fault Tolerance**: The system will implement **retry logic**, **circuit breakers**, and **distributed caching mechanisms** (e.g., Redis or Memcached) to ensure continuous gameplay. Fallback servers or backup nodes will be available to maintain session continuity during regional or node-level outages.

This distributed, service-oriented architecture ensures high availability, fault tolerance, and a smooth user experience across diverse network conditions and geographic regions.

### ****6. Security****

To protect users and data across all supported platforms, Draw It or Lose It employs a multi-layered security model that addresses authentication, encryption, data integrity, and platform-specific concerns:

### ****Authentication & Authorization****

* **OAuth 2.0** is used as the core authentication framework, allowing users to log in via trusted providers (e.g., Google, Apple, or email/password flows).
* **JWT tokens** are issued after login, securely signed to prevent tampering, and stored in platform-specific containers for stateless session management.
* **Role-Based Access Control (RBAC)** governs privileged actions, such as moderating content or accessing analytics dashboards, by assigning roles with granular permissions.

### ****Encryption Protocols****

* **Data In Transit**: All network traffic between client applications and backend services is encrypted using **TLS 1.3**, which reduces handshake latency compared to earlier versions.
* **Data At Rest**: Sensitive user data, like profile information, game history, and scores, is encrypted using **AES-256**, a symmetric encryption standard approved by NIST for securing classified data.

### ****Secure Data Handling****

* **Input Validation & Sanitization**: All input fields undergo strict validation using allow-lists and encoding techniques to prevent:
  + **Cross-Site Scripting (XSS)**
  + **SQL Injection**
  + **Command Injection**
  + **Path Traversal Attacks**
* Validation is performed on both client and server sides to ensure defense in depth.

### ****Additional Security Controls****

* **Rate Limiting & IP Blacklisting**: Protect against brute-force attacks and denial-of-service attempts by limiting the number of allowed requests per client IP.
* **Security Headers on Web Clients**:
  + X-Content-Type-Options: nosniff prevents browsers from interpreting files as a different MIME type.
  + Content-Security-Policy (CSP) defines allowed content sources, helping to mitigate XSS attacks.
  + X-Frame-Options: DENY protects against clickjacking by preventing the game from being embedded in iframes.
* **Regular Security Audits**: Static code analysis, penetration testing, and dependency scanning are conducted during development cycles to detect vulnerabilities early.

This layered approach ensures that user data remains secure from the client app all the way to the backend infrastructure, maintaining user trust and regulatory compliance (e.g., GDPR, COPPA).